# Specification of FlatZinc
## Version 1.1

Ralph Becket

# Contents

# 1 Introduction

This document is the specification of the FlatZinc modelling language.

FlatZinc is the target constraint modelling language into which MiniZinc models are translated. It is a very simple solver independent problem specification language, requiring minimal implementation effort to support.

Throughout this document: $r_1$, $r_2$ denote float literals; $x_1$, $x_2$, $\ldots x_k$, $x_i$, $n$, $i$, $j$, $k$ denote int literals; $y_1$, $y_2$, $\ldots y_k$, $y_i$ denote literal array elements.

# 2 Comments

Comments start with a percent sign, %, and extend to the end of the line. Comments can appear anywhere in a model.

# 3 Types

There are three varieties of types in FlatZinc.

- *Parameter* types apply to fixed values that are specified directly in the model.
- *Variable* types apply to values computed by the solver during search. Every parameter type has a corresponding variable type; the variable type being distinguished by a `var` keyword.
- *Annotations* and *strings*: annotations can appear on variable declarations, constraints, and on the solve goal. They provide information about how a variable or constraint should be treated by the solver (e.g., whether a variable should be output as part of the result or whether a particular constraint should implemented using domain consistency). Strings may appear as arguments to annotations, but nowhere else.

## 3.1 Parameter types

Parameters are fixed quantities explicitly specified in the model (see rule `par_type` in Appendix B).

| | |
|---|---|
| `bool` | — `true` or `false` |
| `float` | — unbounded float |
| $r_1$`..`$r_2$ | — bounded float |
| `int` | — unbounded int |
| $x_1$`..`$x_2$ | — int in range |
| `{`$x_1$`, `$x_2$`, ..., `$x_k$`}` | — int in set |
| `set of int` | — subset of int |
| `set of `$x_1$`..`$x_2$ | — subset of int range |
| `set of {`$x_1$`, `$x_2$`, ...`$x_k$`}` | — subset of int set |
| `array [1..`$n$`] of bool` | — array of bools |
| `array [1..`$n$`] of float` | — array of unbounded floats |
| `array [1..`$n$`] of `$r_1$`..`$r_2$ | — array of floats in range |
| `array [1..`$n$`] of int` | — array of unbounded ints |
| `array [1..`$n$`] of `$x_1$`..`$x_2$ | — array of ints in range |
| `array [1..`$n$`] of set of int` | — array of sets of ints |
| `array [1..`$n$`] of set of `$x_1$`..`$x_2$ | — array of sets of ints in range |
| `array [1..`$n$`] of set of {`$x_1$`, `$x_2$`, ...`$x_k$`}` | — array of subsets of set |

A range $x_1$`..`$x_2$ denotes a closed interval $\{x | x_1 \leq x \leq x_2\}$.

A parameter may be used where a variable is expected, but not vice versa.

An array type appearing in a predicate declaration may use just `int` instead of `1..`$n$ for the array index range in cases where the array argument can be of any length.

## 3.2 Variable types

Variables are quantities decided by the solver (see rule `var_type` in Appendix B).

`var bool`
`var float`
`var `$r_1$`..`$r_2$
`var int`
`var `$x_1$`..`$x_2$
`var {`$x_1$`, `$x_2$`, ..., `$x_k$`}`
`var set of `$x_1$`..`$x_2$
`var set of {`$x_1$`, `$x_2$`, ..., `$x_k$`}`
`array [1..`$n$`] of var bool`
`array [1..`$n$`] of var float`
`array [1..`$n$`] of var `$r_1$`..`$r_2$
`array [1..`$n$`] of var int`
`array [1..`$n$`] of var `$x_1$`..`$x_2$
`array [1..`$n$`] of var set of `$x_1$`..`$x_2$
`array [1..`$n$`] of var set of {`$x_1$`, `$x_2$`, ...`$x_k$`}`

An array type appearing in a predicate declaration may use just `int` instead of `1..`$n$ for the array index range in cases where the array argument can be of any length.

### 3.3 The string type

String literals and literal arrays of string literals can appear as annotation arguments, but not elsewhere. Strings have the same syntax as in C programs (namely, they are delimited by double quotes and the backslash character is used for escape sequences).

### Examples

```
""                  % The empty string.
"Hello."
"Hello,\nWorld"     % A string with an embedded newline.
```

## 4 Values and expressions

(See rule expr in Appendix B).

Examples of literal values:

| Type | Literals |
|---|---|
| bool | true, false |
| float | 2.718, -1.0, 3.0e8 |
| int | -42, 0, 69 |
| set of int | {}, {2, 3, 5}, 1..10 |
| arrays | [], [$y_1$, ..., $y_k$] |

where each array element $y_i$ is either: a non-array literal; the name of a non-array parameter or variable, $v$; or a subscripted array parameter or variable, $v[j]$, where $j$ is an int literal. For example:

```
[1, 2, 3]
[x, y, z]               % Where x, y, and z are variables or parameters.
[a[3], a[2], a[1]]      % Where a is an array variable or parameter.
[x, a[1], 3]
```

Appendix B gives the regular expressions specifying the syntax for float and int literals.

# 5 FlatZinc models

A FlatZinc model consists of:

1. zero or more external predicate declarations (i.e., a non-standard predicate that is supported directly by the target solver);
2. zero or more parameter declarations;
3. zero or more variable declarations;
4. zero or more constraints;
5. a solve goal

in that order.

FlatZinc uses the ASCII character set.

FlatZinc syntax is case sensitive (`foo` and `Foo` are different names). Identifiers start with a letter ([`A-Za-z`]) and are followed by any sequence of letters, digits, or underscores ([`A-Za-z0-9_`]).

The following keywords are reserved and cannot be used as identifiers: `annotation`, `any`, `array`, `bool`, `case`, `constraint`, `diff`, `div`, `else`, `elseif`, `endif`, `enum`, `false`, `float`, `function`, `if`, `in`, `include`, `int`, `intersect`, `let`, `list`, `maximize`, `minimize`, `mod`, `not`, `of`, `satisfy`, `subset`, `superset`, `output`, `par`, `predicate`, `record`, `set`, `solve`, `string`, `symdiff`, `test`, `then`, `true`, `tuple`, `union`, `type`, `var`, `where`, `xor`.
Note that some of these keywords are not used in FlatZinc. They are reserved because they are keywords in Zinc and MiniZinc.

FlatZinc syntax is insensitive to whitespace.

## 5.1 Predicate declarations

(See rule pred_decl in Appendix B.)

Predicates used in the model that are not standard FlatZinc must be declared at the top of a FlatZinc model, before any other lexical items. Predicate declarations take the form

```
predicate predname(type:  argname, ...);
```

where `predname` and `argname` are identifiers.

Annotations are not permitted anywhere in predicate declarations.

It is illegal to supply more than one predicate declaration for a given `predname`.

### Examples

```
% m is the median value of x, y, z.
```

```
    %
predicate median_of_3(var int: x, var int: y, var int: z, var int: m);

    % all_different([x1, .., xn]) iff
    % for all i, j in 1..n: xi != xj.
    %
predicate all_different(array [int] of var int: xs);

    % exactly_one([x1, .., xn]) iff
    % there exists an i in 1..n: xi = true
    % and for all j in 1..n: j != i -> xj = false.
    %
predicate exactly_one(array [int] of var bool: xs);
```

## 5.2 Parameter declarations

(See rule param_decl in Appendix B.)

Parameters have fixed values and must be assigned values:

paramtype: paramname = literal;

where paramtype is a parameter type, paramname is an identifier, and literal is a literal value.

### Examples

```
float: pi = 3.141;
array [1..7] of int: fib = [1, 1, 2, 3, 5, 8, 13];
bool: beer_is_good = true;
```

## 5.3 Variable declarations

(See rule var_decl in Appendix B.)

Variables have variable types and can be declared with optional assignments (some variables are aliases of other variables and, for arrays, it is often convenient to have fixed permutations of other variables). Variables may be declared with zero or more annotations.

vartype: varname [:: annotation]* [ = arrayliteral] ;

where vartype is a variable type, varname is an identifier, annotation is an annotation, and arrayliteral is a literal array value.

### Examples

```
var 0..9: digit;
```

```
var bool: b;
var set of 1..3: s;
var 0.0..1.0: x;
var int: y :: mip;        % 'mip' annotation: y should be a MIP variable.
array [1..3] of var 1..10: a;
array [1..3] of var 1..10: b = [a[3], a[2], a[1]];
```

## 5.4 Constraints

(See rule constraint in Appendix B.)

Constraints take the following form and may include zero or more annotations:

constraint predname(arg, ...) [:: annotation]*;

where predname is a predicate name, annotation is an annotation, and each argument arg is either: a literal value; the name of a parameter or variable, $v$; or a subscripted array parameter or variable, $v[j]$, where $j$ is an int literal.

### Examples

```
constraint int_le(0, x);    % 0 <= x
constraint int_lt(x, y);    % x <  y
constraint int_le(y, 10);   % y <= 10
    % 'domain': use domain consistency for this constraint:
    % 2x + 3y = 10
constraint int_lin_eq([2, 3], [x, y], 10) :: domain;
```

## 5.5 Solve goal

(See rule solve_goal in Appendix B.)

A model should finish with a solve goal, taking one of the following forms:

solve [:: annotation]* satisfy;

(search for any satisfying assignment) or

solve [:: annotation]* minimize objfn;

(search for an assignment minimizing objfn) or

solve [:: annotation]* maximize objfn;

(search for an assignment maximizing objfn) where objfn is either the name of a variable, $v$, or a subscripted array variable, $v[j]$, where $j$ is an int literal.

A solution consists of a complete assignment where all variables in the model have been given a fixed value.

### Examples

```
solve satisfy;      % Find any solution using the default strategy.

solve minimize w;   % Find a solution minimizing w, using the default strategy.

    % First label the variables in xs in the order x[1], x[2], ...
    % trying values in ascending order.
solve :: int_search(xs, input_order, indomain_min, complete)
    satisfy;    % Find any solution.

    % First use first-fail on these variables, splitting domains
    % at each choice point.
solve :: int_search([x, y, z], first_fail, indomain_split, complete)
    maximize x; % Find a solution maximizing x.
```

## 5.6 Annotations

Annotations are optional suggestions to the solver concerning how individual variables and constraints should be handled (e.g., a particular solver may have multiple representations for int variables) and how search should proceed. An implementation is free to ignore any annotations it does not recognise, although it should print a warning on the standard error stream if it does so. Annotations are unordered and idempotent: annotations can be reordered and duplicates can be removed without changing the meaning of the annotations.

An annotation is either

```
annotationname
```

or

```
annotationname(annotationarg, ...)
```

where annotationname is an identifier and annotationarg is any expression (which may also be another annotation — that is, annotations may be nested inside other annotations).

### 5.6.1 Search annotations

While an implementation is free to ignore any or all annotations in a model, it is recommended that implementations at least recognise the following standard annotations for solve goals.

```
seq_search([searchannotation, ...])
```

allows more than one search annotation to be specified in a particular order (otherwise annotations can be handled in any order).

A searchannotation is one of the following:

```
int_search(vars, varchoiceannotation, assignmentannotation, strategyannotation)
bool_search(vars, varchoiceannotation, assignmentannotation, strategyannotation)
set_search(vars, varchoiceannotation, assignmentannotation, strategyannotation)
```

where vars is an array variable name or an array literal specifying the variables to be assigned (ints, bools, or sets respectively).

varchoiceannotation specifies how the next variable to be assigned is chosen at each choice point. Possible choices are as follows (it is recommended that implementations support the starred options):

| | | |
|---|---|---|
| `input_order` | ⋆ | Choose variables in the order they appear in vars. |
| `first_fail` | ⋆ | Choose the variable with the smallest domain. |
| `anti_first_fail` | | Choose the variable with the largest domain. |
| `smallest` | | Choose the variable with the smallest value in its domain. |
| `largest` | | Choose the variable with the largest value in its domain. |
| `occurrence` | | Choose the variable with the largest number of attached constraints. |
| `most_constrained` | | Choose the variable with the smallest domain, breaking ties using the number of constraints. |
| `max_regret` | | Choose the variable with the largest difference between the two smallest values in its domain. |

assignmentannotation specifies how the chosen variable should be constrained. Possible choices are as follows (it is recommended that implementations support the starred options):

| | | |
|---|---|---|
| `indomain_min` | ⋆ | Assign the smallest value in the variable's domain. |
| `indomain_max` | ⋆ | Assign the largest value in the variable's domain. |
| `indomain_middle` | | Assign the value in the variable's domain closest to the mean of its current bounds. |
| `indomain_median` | | Assign the middle value in the variable's domain. |
| `indomain` | | Nondeterministically assign values to the variable in ascending order. |
| `indomain_random` | | Assign a random value from the variable's domain. |
| `indomain_split` | | Bisect the variable's domain, excluding the upper half first. |
| `indomain_reverse_split` | | Bisect the variable's domain, excluding the lower half first. |
| `indomain_interval` | | If the variable's domain consists of several contiguous intervals, reduce the domain to the first interval. Otherwise just split the variable's domain. |

Of course, not all assignment strategies make sense for all search annotations (e.g., `bool_search` and `indomain_split`).

Finally, strategyannotation specifies a search strategy; implementations should at least support `complete` (i.e., exhaustive search).

### 5.6.2 Output annotations

Model output is specified through variable annotations. Non-array output variables should be annotated with `output_var`. Array output variables should be annotated with `output_array([`$x_1$`..`$x_2$`, ...])` where $x_1$`..`$x_2$`, ...` are the index set ranges of the original array (it is assumed that the FlatZinc model was derived from a higher level model written in, say, MiniZinc, where the original array may have had multiple dimensions and/or index sets that do not start at 1).

### 5.6.3 Variable definition annotations

To support solvers capable of exploiting functional relationships, a variable defined as a function of other variables may be annotated thus:

```
var int: x :: is_defined_var;
...
constraint int_plus(y, z, x) :: defines_var(x);
```

(The `defines_var` annotation should appear on exactly one constraint.) This allows a solver to represent `x` internally as a representation of `y`+`z` rather than as a separate constrained variable. The `is_defined_var` annotation on the declaration of `x` provides "early warning" to the solver that such an option is available.

### 5.6.4 Intermediate variables

Intermediate variables introduced during conversion of a higher-level model to FlatZinc may be annotated thus:

```
var int: TMP :: var_is_introduced;
```

This information is potentially useful to the solver's search strategy.

### 5.6.5 Constraint annotations

Annotations can be placed on constraints advising the solver how the constraint should be implemented. Here are some constraint annotations supported by some solvers:

| | |
|---|---|
| `bounds` or `boundsZ` | Use integer bounds propagation. |
| `boundsR` | Use real bounds propagation. |
| `boundsD` | A tighter version of `boundsZ` where support for the bounds must exist. |
| `domain` | Use domain propagation. |
| `priority(k)` | where `k` is an integer constant indicating propagator priority. |

# 6 Output

An implementation should output values for all and only the variables annotated with `output_var` or `output_array` (output annotations must not appear on parameters).

For example:

```
var 1..10: x :: output_var;
var 1..10: y;       % y is not output.
    % Output zs as a "flat" representation of a 2D array:
array [1..4] of var int: zs :: output_array([1..2, 1..2]);
```

All non-error output should be sent to the standard output stream.

Output should be in alphabetical order and take the following form:

varname = literal;

or, for array variables,

varname = `array`$N$`d(`$x_1$`..`$x_2$`, ..., [`$y_1$`, `$y_2$`, ...`$y_k$`]);`

where $N$ is the number of index sets specified in the corresponding `output_array` annotation, $x_1$..$x_2$, ... are the index set ranges, and $y_1$, $y_2$, ... $y_k$ are literals of the element type.

The intention is that the output of a FlatZinc model solution should be suitable for input to a MiniZinc model as a data file (this is why parameters should not be included in the output).

Implementations should ensure that *all* model variables (not just the output variables) have satisfying assignments before printing a solution.

The output for a solution must be terminated with ten consecutive minus signs on a separate line: `----------`.

Multiple solutions may be output, one after the other, as search proceeds.

If at least one solution has been found and search then terminates having explored the whole search space, then ten consecutive equals signs should be printed on a separate line: `==========`.

If no solutions have been found and search terminates having explored the whole search space, then `=====UNSATISFIABLE=====` should be printed on a separate line.

If the objective of an optimization problem is unbounded, then `=====UNBOUNDED=====` should be printed on a separate line.

If no solutions have been found and search terminates having *not* explored the whole search space, then `=====UNKNOWN=====` should be printed on a separate line.

Implementations may output further information about the solution(s), or lack thereof, in the form of FlatZinc comments.

### Examples

Asking for a single solution to this model:

```
var 1..3: x :: output_var;
solve satisfy
```

might produce this output:

```
x = 1;
----------
```

Asking for all solutions to this model:

```
array [1..2] of var 1..3: xs :: output_array([1..2]);
constraint int_lt(xs[1], xs[2]);    % x[1] < x[2].
solve satisfy
```

might produce this output:

```
xs = array1d(1..2, [1, 2]);
----------
xs = array1d(1..2, [1, 3]);
----------
xs = array1d(1..2, [2, 3]);
----------
==========
```

Asking for a single solution to this model:

```
var 1..10: x :: output_var;
solve maximize x;
```

should produce this output:

```
x = 10;
----------
==========
```

The row of equals signs indicates that a complete search was performed and that the last result printed is the optimal solution.

Asking for the first three solutions to this model:

```
var 1..10: x :: output_var;
solve maximize x;
```

might produce this output:

```
x = 1;
----------
x = 2;
----------
x = 3;
----------
```

Because the output does not finish with `==========`, search did not finish, hence these results must be interpreted as approximate solutions to the optimization problem.

Asking for a solution to this model:

```
var 1..3: x :: output_var;
var 4..6: y :: output_var;
constraint int_gt(x, y);    % x > y.
solve satisfy;
```

should produce this output:

```
==========
```

indicating that a complete search was performed and no solutions were found (i.e., the problem is unsatisfiable).

# A Standard FlatZinc Predicates

The type signature of each required predicate is preceded by its specification ($n$ denotes the length of any array arguments).

A target solver is not required to implement the complete set of standard FlatZinc predicates. Solvers are, however, required to support `bool_eq` for all fixed argument values (e.g., model inconsistency detected during flattening may be handled by including a constraint `bool_eq(true, false)` in the FlatZinc model).

$(\forall\ i \in 1..n:\ as[i]) \ \leftrightarrow\ r$ where $n$ is the length of $as$
`array_bool_and(array [int] of var bool: as, var bool: r)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_bool_element(var int: b, array [int] of bool: as, var bool: c)`

$(\exists\ i \in 1..n:\ as[i]) \ \leftrightarrow\ r$ where $n$ is the length of $as$
`array_bool_or(array [int] of var bool: as, var bool: r)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_float_element(var int: b, array [int] of float: as, var float: c)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_int_element(var int: b, array [int] of int: as, var int: c)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_set_element(var int: b, array [int] of set of int: as,  set of int: c)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_var_bool_element(var int: b, array [int] of var bool: as, var bool: c)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_var_float_element(var int: b, array [int] of var float: as, var float: c)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_var_int_element(var int: b, array [int] of var int: as, var int: c)`

$b \in 1..n \ \wedge\ as[b] \ =\ c$ where $n$ is the length of $as$
`array_var_set_element(var int: b, array [int] of var set of int: as,  var set of int: c)`

$(a \ \leftrightarrow\ b = 1) \ \wedge\ (\neg a \ \leftrightarrow\ b = 0)$
`bool2int(var bool: a, var int: b)`

$(a \ \wedge\ b) \ \leftrightarrow\ r$
`bool_and(var bool: a, var bool: b, var bool: r)`

$(\exists\ i \in 1..n_{as}:\ as[i]) \ \vee\ (\exists\ i \in 1..n_{bs}:\ \ \neg bs[i])$ where $n$ is the length of $as$
`bool_clause(array [int] of var bool: as, array [int] of var bool: bs)`

$((\exists\ i \in 1..n_{as}:\ as[i]) \ \vee\ (\exists\ i \in 1..n_{bs}:\ \ \neg bs[i])) \ \leftrightarrow\ r$ where $n$ is the length of $as$
`bool_clause_reif(array [int] of var bool: as, array [int] of var bool: bs,  var bool: r)`

$a = b$
```
bool_eq(var bool: a, var bool: b)
```

$(a = b) \leftrightarrow r$
```
bool_eq_reif(var bool: a, var bool: b, var bool: r)
```

$a \vee \neg b$
```
bool_ge(var bool: a, var bool: b)
```

$(a \vee \neg b) \leftrightarrow r$
```
bool_ge_reif(var bool: a, var bool: b, var bool: r)
```

$a \wedge \neg b$
```
bool_gt(var bool: a, var bool: b)
```

$(a \wedge \neg b) \leftrightarrow r$
```
bool_gt_reif(var bool: a, var bool: b, var bool: r)
```

$\neg a \vee b$
```
bool_le(var bool: a, var bool: b)
```

$(\neg a \vee b) \leftrightarrow r$
```
bool_le_reif(var bool: a, var bool: b, var bool: r)
```

$(a \leftarrow b) \leftrightarrow r$
```
bool_left_imp(var bool: a, var bool: b, var bool: r)
```

$\neg a \wedge b$
```
bool_lt(var bool: a, var bool: b)
```

$(\neg a \wedge b) \leftrightarrow r$
```
bool_lt_reif(var bool: a, var bool: b, var bool: r)
```

$a \neq b$
```
bool_ne(var bool: a, var bool: b)
```

$(a \neq b) \leftrightarrow r$
```
bool_ne_reif(var bool: a, var bool: b, var bool: r)
```

$\neg a = b$
```
bool_not(var bool: a, var bool: b)
```

$(a \vee b) \leftrightarrow r$
```
bool_or(var bool: a, var bool: b, var bool: r)
```

$(a \rightarrow b) \leftrightarrow r$
```
bool_right_imp(var bool: a, var bool: b, var bool: r)
```

$(a \neq b) \leftrightarrow r$
```
bool_xor(var bool: a, var bool: b, var bool: r)
```

$|a| = b$
```
float_abs(var float: a, var float: b)
```

$a = b$
```
float_eq(var float: a, var float: b)
```

$(a = b) \leftrightarrow r$
```
float_eq_reif(var float: a, var float: b, var bool: r)
```

$a \geq b$
```
float_ge(var float: a, var float: b)
```

$(a \geq b) \leftrightarrow r$
```
float_ge_reif(var float: a, var float: b, var bool: r)
```

$a > b$
```
float_gt(var float: a, var float: b)
```

$a \leq b$
```
float_le(var float: a, var float: b)
```

$(a \leq b) \leftrightarrow r$
```
float_le_reif(var float: a, var float: b, var bool: r)
```

$\sum\ i \in 1..n:\ as[i].bs[i] = c$ where $n$ is the common length of $as$ and $bs$
```
float_lin_eq(array [int] of float: as, array [int] of var float: bs, float: c)
```

$(\sum\ i \in 1..n:\ as[i].bs[i] = c) \leftrightarrow r$ where $n$ is the common length of $as$ and $bs$
```
float_lin_eq_reif(array [int] of float: as, array [int] of var float:  bs,
    float: c, var bool: r)
```

$\sum\ i \in 1..n:\ as[i].bs[i] \geq c$ where $n$ is the common length of $as$ and $bs$
```
float_lin_ge(array [int] of float: as, array [int] of var float: bs, float: c)
```

$(\sum\ i \in 1..n:\ as[i].bs[i] \geq c) \leftrightarrow r$ where $n$ is the common length of $as$ and $bs$
```
float_lin_ge_reif(array [int] of float: as, array [int] of var float:  bs,
    float: c, var bool: r)
```

$\sum\ i \in 1..n:\ as[i].bs[i] > c$ where $n$ is the common length of $as$ and $bs$
```
float_lin_gt(array [int] of float: as, array [int] of var float: bs, float: c)
```

$(\sum\ i \in 1..n:\ as[i].bs[i] > c) \leftrightarrow r$ where $n$ is the common length of $as$ and $bs$
```
float_lin_gt_reif(array [int] of float: as, array [int] of var float:  bs,
    float: c, var bool: r)
```

$\sum\ i \in 1..n:\ as[i].bs[i] \leq c$ where $n$ is the common length of $as$ and $bs$
```
float_lin_le(array [int] of float: as, array [int] of var float: bs, float: c)
```

$(\sum\ i \in 1..n:\ as[i].bs[i] \leq c) \leftrightarrow r$ where $n$ is the common length of $as$ and $bs$
```
float_lin_le_reif(array [int] of float: as, array [int] of var float:  bs,
    float: c, var bool: r)
```

$\sum\ i \in 1..n:\ as[i].bs[i] < c$ where $n$ is the common length of $as$ and $bs$
```
float_lin_lt(array [int] of float: as, array [int] of var float: bs, float: c)
```

$(\sum\ i \in 1..n:\ as[i].bs[i]\ <\ c) \leftrightarrow r$ where $n$ is the common length of $as$ and $bs$

```
float_lin_lt_reif(array [int] of float: as, array [int] of var float:  bs,
    float: c, var bool: r)
```

$\sum\ i \in 1..n:\ as[i].bs[i]\ \neq\ c$ where $n$ is the common length of $as$ and $bs$

```
float_lin_ne(array [int] of float: as, array [int] of var float: bs, float: c)
```

$(\sum\ i \in 1..n:\ as[i].bs[i]\ \neq\ c) \leftrightarrow r$ where $n$ is the common length of $as$ and $bs$

```
float_lin_ne_reif(array [int] of float: as, array [int] of var float:  bs,
    float: c, var bool: r)
```

$a\ <\ b$

```
float_lt(var float: a, var float: b)
```

$(a\ <\ b) \leftrightarrow r$

```
float_lt_reif(var float: a, var float: b, var bool: r)
```

$\max(a,\ b)\ =\ c$

```
float_max(var float: a, var float: b, var float: c)
```

$\min(a,\ b)\ =\ c$

```
float_min(var float: a, var float: b, var float: c)
```

$a - b\ =\ c$

```
float_minus(var float: a, var float: b, var float: c)
```

$a\ \neq\ b$

```
float_ne(var float: a, var float: b)
```

$(a\ \neq\ b) \leftrightarrow r$

```
float_ne_reif(var float: a, var float: b, var bool: r)
```

$-a\ =\ b$

```
float_negate(var float: a, var float: b)
```

$a + b\ =\ c$

```
float_plus(var float: a, var float: b, var float: c)
```

$|a|\ =\ b$

```
int_abs(var int: a, var int: b)
```

$a/b\ =\ c$ rounding towards zero.

```
int_div(var int: a, var int: b, var int: c)
```

$a\ =\ b$

```
int_eq(var int: a, var int: b)
```

$(a\ =\ b) \leftrightarrow r$

```
int_eq_reif(var int: a, var int: b, var bool: r)
```

$a\ \geq\ b$

```
int_ge(var int: a, var int: b)
```

$(a \ \geq \ b) \ \leftrightarrow \ r$
`int_ge_reif(var int: a, var int: b, var bool: r)`

$a \ > \ b$
`int_gt(var int: a, var int: b)`

$(a \ > \ b) \ \leftrightarrow \ r$
`int_gt_reif(var int: a, var int: b, var bool: r)`

$a \ \leq \ b$
`int_le(var int: a, var int: b)`

$(a \ \leq \ b) \ \leftrightarrow \ r$
`int_le_reif(var int: a, var int: b, var bool: r)`

$\sum \ i \in 1..n : \ as[i].bs[i] \ = \ c$ where $n$ is the common length of $as$ and $bs$
`int_lin_eq(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum \ i \in 1..n : \ as[i].bs[i] \ = \ c) \ \leftrightarrow \ r$ where $n$ is the common length of $as$ and $bs$
`int_lin_eq_reif(array [int] of int: as, array [int] of var int: bs,      int: c, var bool: r)`

$\sum \ i \in 1..n : \ as[i].bs[i] \ \geq \ c$ where $n$ is the common length of $as$ and $bs$
`int_lin_ge(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum \ i \in 1..n : \ as[i].bs[i] \ \geq \ c) \ \leftrightarrow \ r$ where $n$ is the common length of $as$ and $bs$
`int_lin_ge_reif(array [int] of int: as, array [int] of var int: bs,      int: c, var bool: r)`

$\sum \ i \in 1..n : \ as[i].bs[i] \ > \ c$ where $n$ is the common length of $as$ and $bs$
`int_lin_gt(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum \ i \in 1..n : \ as[i].bs[i] \ > \ c) \ \leftrightarrow \ r$ where $n$ is the common length of $as$ and $bs$
`int_lin_gt_reif(array [int] of int: as, array [int] of var int: bs,      int: c, var bool: r)`

$\sum \ i \in 1..n : \ as[i].bs[i] \ \leq \ c$ where $n$ is the common length of $as$ and $bs$
`int_lin_le(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum \ i \in 1..n : \ as[i].bs[i] \ \leq \ c) \ \leftrightarrow \ r$ where $n$ is the common length of $as$ and $bs$
`int_lin_le_reif(array [int] of int: as, array [int] of var int: bs,      int: c, var bool: r)`

$\sum \ i \in 1..n : \ as[i].bs[i] \ < \ c$ where $n$ is the common length of $as$ and $bs$
`int_lin_lt(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum \ i \in 1..n : \ as[i].bs[i] \ < \ c) \ \leftrightarrow \ r$ where $n$ is the common length of $as$ and $bs$
`int_lin_lt_reif(array [int] of int: as, array [int] of var int: bs,      int: c, var bool: r)`

$\sum \ i \in 1..n : \ as[i].bs[i] \ \neq \ c$ where $n$ is the common length of $as$ and $bs$
`int_lin_ne(array [int] of int: as, array [int] of var int: bs, int: c)`

$(\sum \ i \in 1..n : \ as[i].bs[i] \ \neq \ c) \ \leftrightarrow \ r$ where $n$ is the common length of $as$ and $bs$
`int_lin_ne_reif(array [int] of int: as, array [int] of var int: bs,      int: c, var bool: r)`

$a \ < \ b$
`int_lt(var int: a, var int: b)`

$(a \ < \ b) \ \leftrightarrow \ r$
`int_lt_reif(var int: a, var int: b, var bool: r)`

$\max(a, \ b) \ = \ c$
`int_max(var int: a, var int: b, var int: c)`

$\min(a, \ b) \ = \ c$
`int_min(var int: a, var int: b, var int: c)`

$a - b \ = \ c$
`int_minus(var int: a, var int: b, var int: c)`

$a - x.b \ = \ c$ where $x \ = \ a/b$ rounding towards zero.
`int_mod(var int: a, var int: b, var int: c)`

$a \ \neq \ b$
`int_ne(var int: a, var int: b)`

$(a \ \neq \ b) \ \leftrightarrow \ r$
`int_ne_reif(var int: a, var int: b, var bool: r)`

$-a \ = \ b$
`int_negate(var int: a, var int: b)`

$a + b \ = \ c$
`int_plus(var int: a, var int: b, var int: c)`

$a \times b \ = \ c$
`int_times(var int: a, var int: b, var int: c)`

$a \ = \ b$
`int2float(var int: a, var float: b)`

$|a| \ = \ b$
`set_card(var set of int: a, var int: b)`

$a - b \ = \ c$
`set_diff(var set of int: a, var set of int: b, var set of int: c)`

$a \ = \ b$
`set_eq(var set of int: a, var set of int: b)`

$(a \ = \ b) \ \leftrightarrow \ r$
`set_eq_reif(var set of int: a, var set of int: b, var bool: r)`

$a \ \supseteq \ b \ \vee \ \min(a \bigtriangleup b) \in b$ where $\bigtriangleup$ is symmetric difference
`set_ge(var set of int: a, var set of int: b)`

$a \ \supset \ b \ \vee \ \min(a \bigtriangleup b) \in b$ where $\bigtriangleup$ is symmetric difference
`set_gt(var set of int: a, var set of int: b)`

$a \in b$

```
set_in(var int: a, var set of int: b)
```

$(a \in b) \;\leftrightarrow\; r$

```
set_in_reif(var int: a, var set of int: b, var bool: r)
```

$a \cap b \;=\; c$

```
set_intersect(var set of int: a, var set of int: b, var set of int: c)
```

$a \;\subseteq\; b \;\vee\; \min(a \triangle b) \in a$

```
set_le(var set of int: a, var set of int: b)
```

$a \;\subset\; b \;\vee\; \min(a \triangle b) \in a$

```
set_lt(var set of int: a, var set of int: b)
```

$a \;\neq\; b$

```
set_ne(var set of int: a, var set of int: b)
```

$(a \;\neq\; b) \;\leftrightarrow\; r$

```
set_ne_reif(var set of int: a, var set of int: b, var bool: r)
```

$a \subseteq b$

```
set_subset(var set of int: a, var set of int: b)
```

$(a \subseteq b) \;\leftrightarrow\; r$

```
set_subset_reif(var set of int: a, var set of int: b, var bool: r)
```

$a \supseteq b$

```
set_superset(var set of int: a, var set of int: b)
```

$(a \supseteq b) \;\leftrightarrow\; r$

```
set_superset_reif(var set of int: a, var set of int: b, var bool: r)
```

$a \triangle b \;=\; c$

```
set_symdiff(var set of int: a, var set of int: b, var set of int: c)
```

$a \cup b \;=\; c$

```
set_union(var set of int: a, var set of int: b, var set of int: c)
```

## B FlatZinc Syntax in BNF

We present the syntax of FlatZinc in standard BNF, adopting the following conventions: sans serif xyz indicates a non-terminal; brackets [$e$] indicate $e$ optional; double brackets [[$a - z$]] indicate a character from the given range; the Kleene star $e\star$ indicates a sequence of zero or more repetitions of $e$ ($\star$ binds tighter than other BNF operators); ellipsis $e, \ldots$ indicates a non-empty comma-separated sequence of $e$; alternation $e_1|e_2$ indicates alternatives. Comments appear in italics after a dash. Note that FlatZinc uses the ASCII character set.

```
flatzinc_model ::=   [pred_decl⋆]  [param_decl⋆]  [var_decl⋆]  [constraint⋆]  solve_goal

pred_decl ::= predicate  identifier(pred_param,...);

pred_param ::= type: identifier

type ::= par_type | var_type

par_type ::= bool
     | float
     | float_const..float_const
     | int
     | int_const..int_const
     | {int_const,...}
     | set of int
     | set of int_const..int_const
     | set of {int_const,...}
     | array [index_set] of bool
     | array [index_set] of float
     | array [index_set] of  float_const..float_const
     | array [index_set] of int
     | array [index_set] of int_const..int_const
     | array [index_set] of {int_const,...}
     | array [index_set] of set of int
     | array [index_set] of set of  int_const..int_const
     | array [index_set] of set of {int_const,...}

var_type ::= var bool
     | var float
     | var float_const..float_const
     | var int
     | var int_const..int_const
     | var {int_const,...}
     | var set of int_const..int_const
     | var set of {int_const,...}
     | array [index_set] of var bool
     | array [index_set] of var float
     | array [index_set] of var  float_const..float_const
     | array [index_set] of var int
     | array [index_set] of var  int_const..int_const
     | array [index_set] of var {int_const,...}
     | array [index_set] of var set of  int_const..int_const
     | array [index_set] of var set of  {int_const,...}
```

index_set ::= `1..`int_const | `int`
    --- *`int` is only allowed in predicate declarations.*

expr ::= bool_const | float_const | int_const | set_const
    | identifier | identifier`[`int_const`]` | array_expr
    | annotation | `"...string constant..."`
    --- *Annotation and string expressions are only permitted in annotation arguments.*

identifier ::= $[[A-Za-z]][[A-Za-z0-9\_]]\star$

bool_const ::= `true` | `false`

float_const ::= int_const$[.[[0-9]][[0-9]]\star][[[eE]]$int_const$]$

int_const ::= $[+-][[0-9]][[0-9]]\star$

set_const ::= int_const`..`int_const | $\{$int_const`,...`$\}$

array_expr ::= `[]` | `[`expr`,...]`

param_decl ::= type`:` identifier `=` expr`;`
    --- *Assignments expressions must be constant.*
    --- *Any parameters in assignments must be declared earlier.*

var_decl ::= type`:` identifier annotations [`=` expr]`;`
    --- *Any vars in assignments must be declared earlier.*

constraint ::= `constraint` identifier`(`expr`,...)` annotations`;`

solve_goal ::= `solve` annotations `satisfy;`
    | `solve` annotations `minimize` expr`;`
    | `solve` annotations `maximize` expr`;`
    --- expr *must be a var name or var array element.*

annotations ::= $[$`::` annotation$]\star$
annotation ::= identifier | identifier`(`expr`,...)`
    --- *Whether an identifier is an annotation or a variable name can be identified from its type.*
    --- *FlatZinc does not permit overloading of names.*